

IMPLEMENTATION PROBLEMS
OF
A COMPATIBLE SYSTEM

GÖRAN FRIES

CODEN: LUNFD6/(NFIP-3006)/1-40/(1978)

1978



LUND UNIVERSITY
Department of Computer Sciences



LUNDS UNIVERSITET
INSTITUTIONEN FÖR INFORMATIONSBEHANDLING
SÖLVEGATAN 14 A
223 62 LUND

LUND, NOVEMBER 1978

IMPLEMENTATION PROBLEMS
OF
A COMPATIBLE SYSTEM

GÖRAN FRIES

CODEN: LUNFD6/(NFIP-3006)/1-40/(1978)



This work is supported by the Swedish National Board
for Technical Development. (75-3354, 76-3609, 77-4426).



C O N T E N T S

	ABSTRACT	1
1.	INTRODUCTION	2
2.	GENERAL DESIGN OF THE LANGUAGE	3
3.	DATA AND MEMORY ORGANIZATION	5
3.1	PRIMARY MEMORY	5
3.2	REGISTERS	6
4.	INSTRUCTIONS	9
4.1	ORDINARY ASSEMBLY INSTRUCTIONS	9
4.2	HIGHER ASSEMBLY INSTRUCTIONS	15
4.3	OPERATING SYSTEM INTERFACE INSTRUCTIONS	18
4.4	GENERAL CONCEPTS	21
5.	THE TEST IMPLEMENTATION ON UNIVAC	25
5.1	THE LEXICAL ANALYZER	25
5.2	THE SEGMENT TRANSLATOR	26
5.3	IMPLEMENTATION PROBLEMS	30
	EXAMPLES	33



Digitized by the Internet Archive
in 2026

ABSTRACT

This report contains some comments on the OBM language[1]. Some ideas behind the language definition are given and some implementation problems are discussed.

1.

INTRODUCTION

The purpose of this report is to discuss problems in implementing a machine independent machine oriented language aiming at compatible systems and to give some suggestions of how to implement such a language. The language, OBM, is defined in [1].

This report will give suggestions of how to implement some of the OBM features, but also why the language is defined in the way it is will be discussed. During the definition of the language a partial test implementation on a UNIVAC 1108 has been made. This report is influenced by this implementation, but it is not a technical report on this implementation. The question: Why are the OBM instructions defined as they are, is important. The answers may be grouped into three classes.

- i/ To solve the compatibility problem this is the best solution. It is a real solution, and it is the simplest one of the solutions to implement.
- ii/ The choice is not really important and this one is a possible and simple solution, and one must be choosen.
- iii/ It just happened to be like this. I have not really understood the problem, and I have not thought of it.

There is nothing peculiar in the two first classes, but what about the third one? In most programming languages the third way of defining concepts is rather common, but it is never mentioned. This is a way of defining some concepts of a language just by chance, and this may give poor definitions. The third class should of course be empty, but I must admit that some of the definitions in OBM belongs to this class. Of course I do not know which ones, and that's the problem. These definitions will not be further described, as I know nothing about them.

The first class is the most interesting one, some of the definitions here are obvious but others are hard to understand. Those non obvious definitions should be the main concern of this report.

The implementation of some choosen concepts of OBM will be discussed. I also hope to prove that the concepts can be implemented in a reasonable efficient way.

2.

GENERAL DESIGN OF THE LANGUAGE

The first question to answer was: Shall the language (OBM) be a high level language or an assembly language? As the language shall be used as an intermediate language between a high level language and machine code, it should be somewhere in between. As OBM will be used to represent programs written in a number of different high level languages, OBM should not be on a level too close to any high level language. On the other hand an OBM program will be executed on a number of different machines, thus OBM should not be on a level too close to the machine code for some computer. If OBM is too close to some computer, it will be an efficient language for that computer, but hard or impossible to implement on other computers. At least in an efficient way.

OBM should be a machine oriented language rather close to the machine level, but distant enough from the machine level to be suitable as an assembly language for a number of quite different machines. OBM should also be able to represent the result of compilation of high level languages. OBM should contain primitive concepts that may be used to represent all different high level languages in an efficient way.

I have defined OBM as a language on a slightly higher level than ordinary assembly languages, but it is much closer to the assembly level than to the high language level.

Another question is about the graphic representation of the language. A standard alphabete should be used, and to facilitate the implementation a common subset should be picked out. For OBM a subset from ISO 646 is used. The subset is such that it is also present in national versions of ISO 646. The graphic representation should also be media independent. That means that lines, card formats, disk and tape format are not significant. A carriage return, line feed or form feed is equivalent to space.

A third question is: What is a program? Can it be divided physically and logically in smaller units? As OBM is a language for a number of different machines, small or big, a program should be built from smaller parts and then linked to a program with a memory allocation suitable for the actual host computer. A program is a normal unit for execution, but not for symbolic code. For symbolic code the segment is a natural unit, and a sequence of segments may be translated as a sort of unit.

2.

A segment may be made small enough to be a suitable building block for a program for any computer. A segment can be regarded as a sequence of statements, this is natural for a machine oriented language.

3. DATA AND MEMORY ORGANIZATION

3.1 PRIMARY MEMORY

In a program it must be possible to allocate a memory area for constants and variables. Most memories are organized as a sequence of units (byte, word) of a certain length in bits. This is dependent on the access width of the hardware. As the memory is a sequence of units of the same length such a memory has a homogenous structure. In word oriented computers there may be instructions handling multiple words and in byte oriented there are instructions handling multiple bytes, but still the memory structure is uncomplicated. The length of the units is a hardware dependent constant, one for each computer make (or model).

The result and performance of a program is of course strongly dependent on this memory structure, as long as the instructions are used in the normal way. To be able to write truly machine independent programs, it must be possible to prescribe the memory structure in the program. The memory structures and "word lengths" are defined in OBM by means of a TYPE statement. Existing computers have one memory structure each, but in OBM it is possible to have many different structures, even in the same program. Is this necessary? Perhaps it is not, but it adds a great deal of flexibility to the language. The problem is to implement one structure at the time foreign to the host computer, and to implement more than one such structure does not make the problem significantly harder.

How shall the, in an OBM program, prescribed foreign structure be forced upon the memory structure of the host computer? For efficiency reasons this may be expressed within the OBM language by the store option of the TYPE statement.

The normal method (N option) is that the significance is completely prescribed, but the storage is left to the implementer. In the next method (W) the significance is still prescribed, but the storage must be in complete words of the host computer. It is also possible (S) to prescribe both significance and storage in a number of complete words. This gives at least specified significance and only portability, but an increase in efficiency.

3.1

The "exact" storage and significance (E) gives a possibility to long bitstring memories specially useful for data exchange with other computers directly or through secondary memories like disks and tapes. The last method may be very expensive to use.

Why do we have restrictions on word lengths, and why are the restrictions chosen as they are? As it must be possible to construct a translator of the language with reasonable size and efficiency a restriction is needed. A maximum word length of 255 bits is regarded as more than enough and the bit length may be represented in the translators type and symbol tables as one byte of information. Other restrictions are more obvious. Perhaps the restriction of character size to 8 bits (one byte) is serious. It gives a possibility to represent a code alphabete of 256 different characters, and this is enough for existing alphabetes. It has been hard to represent more characters on line printers and terminals, but today we have microprogrammed matrix printers with a possibility to represent a large number of different characters and control functions.

In OBM there is also a restriction on memory (pool) size. The restriction is made to limit the address length in bits, and to make it possible to store the pool in the physical memory without any form of paging. Therefore there is, in OBM, both a restriction on the number of words (addresses) in the pool and the number of bits (memory size) of the pool.

These restrictions can be serious for programs using large data arrays, but even if the permitted OBM memory size would be increased the physical memory size would still be the same.

3.2

REGISTERS

Most computers have a special fast and small (short addresses) type of memory called registers. The number of registers and the use of them differs very much from one computer to another. It is very common that at least one of the operands, for some computers both, to an arithmetical instruction must reside in a register, and that the result will be in this register.

3.2

As the address of a register is short this will make it possible to keep instruction length down. How many registers should we have in the flexible OBM languages? What should be the size of the OBM registers?

As a register must be able to hold an operand there must be registers of any permitted size for OBM words. Such registers are hard to implement in an efficient way.

The question: How many registers, can be answered in a number of ways?

(a) One OBM register.

This is easy to implement on most computers, but it may force the programmer to perform many unnecessary data transfers to and from this register. It is also hard to use the full strength of computers having several registers.

(b) A small number of registers.

Any chosen number of registers (2-8) is too large for some computers and too small for others. It is harder to implement than the first suggestion (a), but it weakens the drawback in (a).

(c) A large number of registers.

Large means any number of registers as long as it is not more than an ordinary primary memory. This is rather hard to implement, but unnecessary data transfers are kept to a minimum. It is the responsibility of the OBM translator to use hardware registers or memory cells as OBM registers in an efficient way.

(d) No registers.

This seems to be the opposite of (c), but it is not. It is almost the same thing to implement. The main difference is that in (c) the user has a form of (pseudo) control of the data transfers to and from the computers hardware registers. In (d) this control is completely left to the OBM translator. Method (d) may cause sequences of store and load the same data into the same register more frequently than method (c). This is rather easy to avoid by a simple form of optimization.

3.2

Which one to choose in OBM?

The real candidates are (c) and (d). (c) seems to be slightly better than (d), specially in efficient hardware register usage. During implementation on a UNIVAC 1108 with 16 accumulator registers we discovered that it was too few registers to allow much user control of them and the register usage with (c) and (d) was almost the same. Then it may be noticed that the flexibility in length of OBM registers and the lack of this flexibility of the hardware registers made it very hard to implement the OBM registers in an efficient way. Because of this it was decided to have no registers in OBM, all operands are fetched from memory. This caused OBM to be a three address computer with arithmetical instructions of the complexity $a:=b+c$, where a, b and c are memory locations.

4. INSTRUCTIONS

As discussed in 2 the language is designed to be on the assembly level, and with instructions of a rather uncomplicated kind rather than high level statements. In the report defining OBM [1], the instructions are divided into three categories. These are:

- i/ Ordinary assembly instructions.
- ii/ Higher assembly instructions.
- iii/ Operating system interface instructions.

4.1 ORDINARY ASSEMBLY INSTRUCTIONS

This class of instructions is present in all assembly languages, but the OBM instructions are on a slightly higher level than in other such languages. The reason for this is that an OBM instruction must not be dependent on a certain computer architecture. In this class instructions are included for data transfer, arithmetic, logic and branching. As a first example we may look upon a simple data transfer. As OBM does not contain registers (see 3.2), the normal transfer of operands from memory to a register is not present in OBM programs. If we want to save the contents of one location somewhere else, a simple data transfer is needed. The data transfer is a two address instruction

```
= a,b;
```

with the meaning $a:=b$. This looks very simple, but we have to consider the flexibility of OBM operands. Should there be any restrictions on a and b ? If we could guarantee a and b to have the same word length always, the transfer would be simple. Such a restriction is much too strong to permit efficient programming in OBM. Any form of mixed type operands are permitted in OBM. The transfer can then be described as: take the operand b and change it to fit location a and put it here.

4.1

On UNIVAC the transfer instruction may be implemented as:

- i/ transfer operand b to a register (or two)
- ii/ trim (truncate or expend) the operand to have the same length as a.
- iii/ store the result in a.

For operands with a length less than or equal to 72 bits this is easy on UNIVAC, but for longer words the above sequence is more complicated.

Here no type conversion was included, but if such a conversion is desired the following form of the transfer statement is used:

$$= a, b, t_1, t_2;$$

where t_1 is a type defined in a type statement.

This transfer and conversion may be described as:

Take operand b and convert it from type t_1 to type t_2 , then change the length to fit location a and put it there.

Let the arithmetical instructions be represented by the add instruction:

$$+, t \quad a, b, c;$$

4.1

This is of normal assembly type and the type of arithmetic is completely determined from the operator + and the type t, independent of the operands. This is an important assembly level property, that all data is interpreted under full program control. The operands a, b and c may have different length and the instruction can be described as (see also 5.):

- i/ load b into a register
- ii/ trim b to the length of t.
- iii/ eventually load c into a register
 and trim it to the length of t.
- iv/ add the operands according to type t.
- v/ trim the result to be length of a.
- vi/ store the result in a.

This is complicated and it looks inefficient. Is this complexity necessary? If we permit data of different length in a program, we must be able to make arithmetical operations on such data. This possibility makes the arithmetic slow only when it is used. In good programs most of the operands will be of the same length and the above sequence will be reduced to a few machine instructions. It has to be remembered that much of the inefficiency of OBM is only introduced when the programmer uses the full possibilities of OBM.

The trim function is an important part of OBM. When data of one length is stored in a location with another length or operands of different lengths are used in arithmetical instructions, something has to be done. How to define this trim function? If the data is just to be moved it is impossible to know if the data will be used as an integer or as a real number or as something else.

4.1

The moved data does not have the usage (integer, real, ...) as a property, and this is a normal and important assembly level feature. If data is used as an operand for arithmetic the type of the resultant operand after trim is known, but we have also to know the complete type of the data before trimming. But data does not have a complete type as property!

It is rather easy to trim an integer in a reasonable way, but very hard to trim a floating-point number. Therefor all data are trimmed as if they were to be interpreted as integers. If the word is to be trimmed to a shorter length it is truncated and the high order bits are discarded. For integers with complementary arithmetic the trimmed result will be correct as far as it is possible to truncate it correctly. If the operand is to be expanded, it is expanded to the left with signbits. If the operand is located in a pool defined by means of an integer type with signbit and no complement, the expansion is done by moving the signbit and inserting zeroes. This is not really according to the OBM principles, because a data word does not have the signbit property, but it is welldefined and it gives correct results also for signbit integers. If an operand to real number arithmetic is trimmed the result will be nonsens, but welldefined. Is it then not possible to use mixed type arithmetic with reals? A conversion may be performed under program control by means of the transfer with conversion statement. This is the normal way of doing it at the assembly level.

Types of arithmetic.

What types of arithmetic should be included in OBM? It is quite obvious that integer and real arithmetic of different word lengths have to be included. But it is not only the word length that is important, also signrepresentation, truncation methods and word disposition are very important. For integers it is rather obvious that one's and two's complementary arithmetic, sign bit arithmetic and decimal arithmetic should be included. For floating-point arithmetic it is much more complicated. We may choose sign representation both for exponent and mantissa, we may choose different ways of truncation and we may place exponent and mantissa in different ways in the data word.

4.1

If all combinations of these choices would be permitted in OBM, hundreds of different types, besides word length, of floating-point arithmetic would have to be included. Some of these combinations would never be used. In order to avoid this, only a very limited number of common combinations has been included in OBM.

Are there any problems in implementing these arithmetical instructions? It is of course one of the hard tasks in implementing OBM. Some investigations have been made for UNIVAC 1108. Arithmetic, with word lengths greater than the size of the largest number that can be operated on by existing machine instructions, will be rather slow.

For floating-point numbers the same is true for both the exponent and the mantissa. For other types it is possible to use the existing operations and to make some transformation of the operands and the result. The simulation of two's complementary arithmetic on a computer using one's complement has shown some undesired effects. The absolute value of such a number may be negative, due to the unsymmetric number distribution around zero. This must be checked for, and an overflow signal must be given when it happens. This is not a hard problem, but time consuming.

Among the arithmetical instructions are also the shift instruction included. The logical shifts are rather uncomplicated, at least for reasonable word lengths. The circular shifts are far from simple. If an OBM word of a word length different from that of the host computer, bits are shifted out from the middle of the word and those bits have to reappear at the end, for circular left shifts. Right shifts are not simpler. This is very hard to implement in an efficient way.

The bitwise logical instructions are simple to implement. The "mask and pack" and the "mask and unpack" instructions are convenient for the programmer, but they may be hard to implement in an efficient way.

4.1

Branching.

The conditional branch instructions are rather simple. They are a bit more powerful than the test and skip instructions in most assembly languages. This OBM instructions have been implemented on UNIVAC and the generated code was efficient. The statement structure seems to be appropriate for representing conditional statements in high level languages.

The unconditional branching in OBM has been expanded and made more powerful than just a normal assembly jump instruction. First it must be noticed that it is impossible to talk about the size in bits or computer words of an OBM instruction. The instruction part of the memory is highly inhomogenous, but almost all statements may be labelled. This means that it is impossible to talk about the address of an instruction in any other terms than in terms of the label of an instruction. It is impossible to calculate an address during execution in the normal way. Sometimes it is convenient to have a sort of dynamic branch statement, this is implemented in OBM by the label variable feature. This is a variable that might be assigned a label as value. It is possible to jump to this variable label. This may be implemented by an indirect jump. It is also possible to define a sort of switch by means of the CASE statement. The CASE statement together with the CASEEND statement can be used in a much more powerful way than just a normal switch. Together with the extended GO statement this may be used as a sort of flexible subroutine with multiple entry points. Look at the following example.

```

SUB:CASE L1,L2,L3,L4,L5;
L1:  ~
L2:  ~
L3:  ~
CASEEND;

L4:  ~
L5:  ~
CASEEND;
GO SUB(2),R1;  GO SUB(1),R2;
```


4.1

Example of the CASE as flexible subroutines.

In this example the case may be regarded as two subroutines. The first with entry points L1, L2 and L3, and the second with entry points L4 and L5. These are subroutines in the assembly sense, they are not a closed part of the code as in high level languages. It is possible to enter and leave the code by any OBM branch instruction, but the programmer may use them as subroutines, if they are programmed as subroutines. The unconditional branch instruction GO may also be used to save a return address, that is used by CASEND. No recursion is built into the language.

These flexible branching possibilities built into the language by the CASE, CASEND, LABEL and GO statements are easy to implement. This implementation has been done and tested for UNIVAC.

4.2

HIGHER ASSEMBLY INSTRUCTIONS

These instructions are not usually found in assembly languages, but they are convenient for the programmer and good to represent some high level language concepts. These instructions are not really necessary, they can be replaced by a sequence of other OBM instructions. It is, however, easier to translate one high level instruction to efficient code, than to translate the corresponding sequence of low level instructions. Included higher instructions are loops, stack and string handling and subroutines.

Loops.

OBM contains statements to define special repeat blocks. These blocks are regular loops that are in some respects closed from the outside world. There are two sort of loops, with or without control variable. The loop without control variable is the simplest, and just made to repeat the code a given number of time. This number is determined at the start of the loop and cannot be changed from the loop block. The other form of loop is of "step until" type. The start, step and end value is determined at the start of the loop. If these are variables they might be changed from the loop block, but this has no effect on the loop control.

4.2

It is forbidden to change the control variable in any other way than by the loop control (step) in the repetition. All loop blocks are closed for jumps, in both directions. They are only entered at the beginning repeat statement and left at the end of the loop, or through a special EXIT statement. When a loop is left the next statement to execute is always the following statement after the loop block (REPEND).

The reason to introduce such regular loops in OBM is that the loop control can be more efficiently coded than if the loop is programmed in OBM low level instructions. As the loop is very regular, all index checking, for the control variable as index, in the loop block may be made at the entrance of the loop. The result of the text implementation is that the loop can be efficiently implemented in a rather simple way. But it is also possible to optimize the loop control in a far from simple way, this has not been done in the test implementation. For loop blocks and scope of variables and labels, see also 4.4.

Stacks and strings.

OBM is supposed to be used also for compiler construction and other system or system oriented programming. Such programs are often using strings and stacks. The control of strings and stacks can be made more efficient if special OBM instructions for this are included. It is easy to define a stack, but it is defined to reside in the normally defined pools, and no protection against other use of the stack area is included. Remember that it is still assembly level programming! As long as the stack area is manipulated with the special stack instructions there is an automatic protection against under- and overflow. The stack feature is rather easy to implement.

It is possible to define a string in a special predefined string pool, and to manipulate this by the string instructions of OBM. In order to be efficient the strings are stored in a machine dependent way, but as long as they are used internally by the OBM program and only manipulated by the string instructions, this machine dependence is invisible to the programmer. As strings have machine dependent representation they are not appropriate for data exchange with other machines through secondary memories like disks and tape.

4.2

There are special instructions, PACK and UNPACK, for conversion between strings and machine independently stored data. Strings are also operated on by I/O instructions, see 4.3. Also strings can be implemented in a rather unproblematic way.

Subroutines.

We have already met assembly level "subroutines" in 4.1 (CASE), but also the high level language type of subroutines are included in OBM. They are included mainly as building blocks in libraries, and they will very often be translated completely independent of the calling program. An OBM subroutine can be regarded as a block of code, with a name, that is closed from the outside world. The block of code may only be entered by a subroutine call and only left by executing a return instruction. Data are transferred between the routine and the calling program through parameters. Also global (external) variables are permitted.

The greatest problem in implementing subroutines is the parameter transfer. The problem is in some respects the same as for a high level language like Algol or Simula, but the flexible word lengths and storage makes the problem much harder.

A subroutine with, say, a 16 bit integer as parameter, should be able to operate on this regardless of how it is stored. The same subroutine should be used if the integer is stored as 16 bits in a full computer word or as 16 bits in a half-word, packed with other such integers or as a 16 bit string in memory regardless of word boundaries. Perhaps the integer is stored as a partial word in a node. The pools where it is stored can also be of different size, and that is important for index checking.

All these things make it impossible for the subroutine to address the parameter. All addressing of a parameter has to be done in the calling program. The subroutine knows nothing about the storage of its actual parameters, but it knows the length in bits of the parameters, as this is declared in the subroutine description. The parameters may have different type of usage, value, reference or absolute number. The last is simple, just a constant as actual parameter. For value parameters the transfer can take place at the call of the subroutine, and this is rather simple.

4.2

For reference parameters, the parameter may be referenced for loading and storing at any stage of the subroutine execution. The addressing code (trim included) for both load and store must be generated in the calling program and the two addresses of the addressing routines are transferred to the routine at entrance. The parameter transfer is obviously possible to implement in a complicated but not too inefficient way. If we regard the subroutine as a form of user defined OBM instruction, we may see the similarities between a subroutine call and the use of an instruction.

We have also different types of subroutines, the simplest is the non-recursive subroutine. This is rather simple to implement, but it may be hard to check if such a routine is used in a recursion. Also recursive subroutines are included. In a recursive subroutine a block is pushed on a special subroutine stack at the call. This block contains a return address, the value of the parameters and a specially declared local work space. The call of such a routine is of course more expensive than the call of a non-recursive one. An efficient implementation of this stack and its administration may cause some problems.

For efficiency reasons and to be able to use some special machine properties, also external, often assembly written, subroutines are included. These are not for general use, but intended for system programmers and experts knowing the machine and the OBM implementation.

Such a routine can be regarded as something outside of the program and the program takes no responsibility of what happens at the outside.

4.3

OPERATING SYSTEM INTERFACE INSTRUCTIONS

The machine instructions of most computers can be divided into two classes, and the machine can operate in two modes, privileged and user mode. In privileged mode all instructions may be executed but in user mode the execution of some instructions will cause a sort of error interrupt.

4.3

The instructions only executable from privileged mode, privileged instructions, are those instructions that may be dangerous in a multi-user environment. The service of the privileged instructions is given the user through the operating system. These operating system service routines are included in OBM as normal instructions. These instructions can be divided into I/O instructions and execution control instructions.

I/O.

I/O is regarded as a transfer of bits (data) between the primary memory and some secondary memory. There are two I/O instructions in OBM, READ and WRITE. The transfer is between a buffer in primary memory and a secondary memory area. The secondary memories are of different types, disks, tapes, printers, card readers, terminals, etc. Notice that there is no real difference between the transfer of some bits to a disk and some bits to a line printer. What will happen to the data in the secondary memory is of course device dependent. Data transferred to a line printer will be printed as characters on paper (or control printer action), but data transferred to a disk device will be stored on the magnetic surface. This makes it simple. If data representing some text, is to be saved on a disk or printed on a line printer, the same instruction may be used and only the device name has to be changed. Secondary memories are organized in items, as the smallest addressable unit. There are then two different ways of organizing the items, in characters or in words of a given bit length. I/O is a transfer of bits, but for character organized devices a translation of characters from one code to another may be performed during I/O. Even if an I/O instruction is the same for all devices we have to consider the special device properties. It is impossible to read from a line printer, to randomly access an item on a sequential device like a tape.

Most computers permit the CPU to work during data transfers to and from secondary devices. To be able to use this within a program, it is possible to tell the system if the program wants to wait for the transfer to be finished or to have the control back immediately. It is also possible to exclusively assign a shared device for a critical part of the program. All this is done by putting an option on the I/O statement.

4.3

It must be noticed that if an I/O statement is legal or not depends on the properties of the addressed secondary device. These properties are defined in the operating system, as fixed standard properties (as for line printers) or by commands to the operating system.

Execution control.

Sometimes it is useful to start the execution of a program from another program. It is also useful to construct a system of parallel programs and to synchronize them. This is specially useful if a multi processor system is available. To be able to use this style of programming, such execution control statements are included in OBM.

The simplest form of execution control is to start an independent program. This may be useful to do from an interactive program at certain points. This is of course very simple to implement, it has to be present in the operating system anyhow.

An activity may be regarded as a piece of code and data and an execution path (sequence control) in this code. An executing program is then a number of activities run in parallel and with possibilities of synchronization and communication. In a multi-processor system true parallelism is possible, else it has to be simulated by the operating system.

There are two sorts of parallelism between activities. The first is that the two activities are different execution paths in the same code. Here all protection between activities must be handled by the programmer, and it is only the sequence control that is parallel, not the code and data. This is rather simple to implement and it is also the sort of parallelism existing in the operating system of UNIVAC. The second form of parallelism is that not only the sequence control, but also the code and data are "parallel". Here it is impossible for one activity to access code and data in another activity. All communication between activities has to be done by special synchronizing statements or through a common data area.

4.3

This data area can only be accessed by one activity at the time, the area is assigned to one of the activities and thus protected against other activities. It is assigned to an activity until released by this activity. This makes it impossible for other activities to change the value of flags and data areas between the test of flags and processing data depending on that test.

There are also synchronization commands. It is possible for an activity to wait until some other named activity is terminated. It is also possible to wait until another activity gives a special activation signal. It is possible to activate another activity. These possibilities can be used to write programs where routines and parts of the code are executed in parallel and at certain points wait until the other activities has reached certain points in their execution.

This second form of parallelism is more complicated than the first, but also more useful. An example of where this possibility could be used is interactive library routine usage. An interactive program is used for problemsolving, say, in engineering. The program asks questions about the problem, chooses a routine and executes it and goes on asking more questions. This can be made more efficient in a system where routines can be run in parallel and synchronized at certain points.

4.4

GENERAL CONCEPTS

Some important general concepts have to be defined in an unambiguous way. These are a) addressing, b) scope of definitions and c) operations on operands of different word length.

Addressing

Data words can be addressed by a name, this is convenient and rather simple to implement. Also addressing by name and a numerical index (offset) may be used. Here all address computation and checking can be done during translation, and this is just as simple as addressing without index.

4.3

This data area can only be accessed by one activity at the time, the area is assigned to one of the activities and thus protected against other activities. It is assigned to an activity until released by this activity. This makes it impossible for other activities to change the value of flags and data areas between the test of flags and processing data depending on that test.

There are also synchronization commands. It is possible for an activity to wait until some other named activity is terminated. It is also possible to wait until another activity gives a special activation signal. It is possible to activate another activity. These possibilities can be used to write programs where routines and parts of the code are executed in parallel and at certain points wait until the other activities has reached certain points in their execution.

This second form of parallelism is more complicated than the first, but also more useful. An example of where this possibility could be used is interactive library routine usage. An interactive program is used for problemsolving, say, in engineering. The program asks questions about the problem, chooses a routine and executes it and goes on asking more questions. This can be made more efficient in a system where routines can be run in parallel and synchronized at certain points.

4.4

GENERAL CONCEPTS

Some important general concepts have to be defined in an unambiguous way. These are a) addressing, b) scope of definitions and c) operations on operands of different word length.

Addressing

Data words can be addressed by a name, this is convenient and rather simple to implement. Also addressing by name and a numerical index (offset) may be used. Here all address computation and checking can be done during translation, and this is just as simple as addressing without index.

4.4

To be able to use more flexible addressing, variable indexing has to be introduced. This will give slower object code and translation, because of index checking during runtime and complicated generation of load instructions in object code. As the pool size is limited, large indices are nonsens. An index with more than 16 significant bits (other than sign-bits) is always an error. In order to facilitate implementation, a data word with a type containing more than 64 bits is forbidden, and only 16 bits are significant. An index must not be indexed, and for signrepresentation and word disposition of an index, the type used to define the index must not prescribe decimal arithmetic.

Security is an important feature of OBM. Numerical indexing is checked at translation time and variable indexing is checked during run time. The run time check is time consuming and therefore some tools to avoid this are provided. In a repeat block (REPEAT ----- REPEND) some index checks can be done at the entry of the repeat block. If this is done in a bad way a possible index out of range will give an error interrupt even if the repeat block will be left before the real error, by the EXIT statement. Thus if an EXIT statement is present care must be taken when trying to optimize the error checking.

Another way to avoid error checking of index range, is to inhibit the generation of object code for this checking. This may be done for the whole program or a part of the code, by the pseudo instruction INDCH. If this is done the security is (partially) sacrificed, and the program may run into an uncontrolled state. For some production programs the efficiency is so important that index checking should be inhibited. But this should be done only after the program has been well tested, with index checking turned on.

For structured data words, two indices will be used. What is said above about indexing is also true for the two indices on a structured word. The efficiency problems are here at least twice as great, if both indices are variable.

Trimming is already described in 4.1.

Scope and block structure.

OBM is an assembly language and does not have a block structure in the Algol sense. An OBM segment cannot be regarded as a block, but it may contain block-like units. One form of block is the closed subroutine, such blocks cannot be nested. Subroutine blocks are often translated separated from the calling program and the block property is not obvious. Another form of blocks are the repeat blocks, those are more similar to Algol blocks.

As the loops are desired to be regular and disciplined, such a block is closed for jumps in both directions. Such blocks may be nested to the depth of eight.

The normal scope of data words, labels and other defined symbols is the segment. For communication between segments it is possible to declare some symbols to be referencable from other segments or that references are made to other segments. Such symbols are called external. Defined subroutines are always external, and they must always be defined at the top level of a segment, not within a block of any kind. Data words are always global in the segment and they can be declared to be external. External data words have to be defined, and completely described, in all segments using them. This problem is not present in normal compilers and assemblers, as the memory structure and data format is evident. In OBM we do not have such an evident structure, and the structure can be different in different segments of a program.

If a data word would be described only in the segment where it is allocated, then other segments using them cannot produce object code for their loading. To produce this code during the linking of segments is not a good method. As they are completely described in all segments, the linker can just check if the types are compatible and thus the produced code for data references is correct in all segments.

Instruction labels are also global in the segment, if they are not defined within a block. Such labels can be declared external. Also external references have to be declared, this is to simplify the translator and linker.

4.4

Label variables and switch labels are global in a segment if they are defined outside of all blocks. They cannot be declared external. A global variable label or switch may have values that are global or external, but never local. If local values would be permitted for a global label variable it would be very hard to check for illegal branching into blocks.

The names of types and pools are just used during translation, and they have no external meaning. They are always defined at the top level of a segment.

The block structure makes it possible to define local symbols. An instruction label is strictly local and only referenceable on the block level where it is defined. This is also true for label variables and switches, and they can only hold local values. This is to guarantee the blocks to be closed for branching in both directions. This disciplined branching makes optimization of the object code easier. Data words in pools can only be defined as global, and never local. In repeat blocks it is possible to have local variables, but only the control variable and secondary variables whose values are a function of the control variable. The value of these local variables can only be changed by the loop control (the step), but their value can be used as indices and operands in ordinary statements. What happens if a local variable has the same name as a global, or a local on another level? If the variable is referenced in order to use its value, there are no problems, the same rule as in Algol is used. If the local variable is used as a receiving variable or if it is indexed, it is an error. This error has to be detected during translation and if a global variable with the same name exists, this variable is used. But remember it is an error!

The subroutine block is somewhat different from the repeat block. Instruction labels in a subroutine is handled as those in a repeat block. Local data words can be defined in a subroutine. In a recursive subroutine such data words can be defined by the LOCAL statement. Parameters to a subroutine described as value, will also cause a local variable with this value (at start) to be created. Local variables in a subroutine may be changed in the same way as other data words.

5.

THE TEST IMPLEMENTATION ON UNIVAC

A detailed technical description will not be given here, but a description of the general design that might be helpful when implementing the OBM language somewhere. The translator is divided into two parts, the first is a lexical analyser and the second is the real translator. The lexical analyzer is a pass operating on an OBM program, that is a sequence of symbolic segments. To the first pass the OBM program is just a character string according to the rules of OBM. The second part is a two pass translator that is repeatedly operating on the different segments.

5.1

THE LEXICAL ANALYZER

This is written as a modified pushdown automata (or rather transducer). A special simple automata language has been constructed and used for writing the lexical analyzer. This automata language is implemented by means of a LISP program that translates the statements in the automata language to assembly code. The produced code for the lexical analyzer is rather efficient, but it requires rather much space. But compared to the rest of the translator it is small. One statement at the time is translated and the result is represented in a 14 word table, for some statements this table can be extended by one or more 14 word units. The result after the first pass (the lexical analyzer) is a sequence of such tables. It is too much to describe the design of these tables for all statements, but one typical statement will be described.

A table for a statement

{LABEL:}OP{,SPEC} -----;

where

label is LABEL or 0

n= { 0 last 14 word unit for statement
1 statement will continue in the next
14 word unit

op= { a code for OP, operators numbered 1,2,---
0 if a continuation unit

spec is SPEC or 0

label	
n	op
spec	
P	
a	
r	
a	
m	
e	
t	
e	
r	
s	

5.1

LAB	0000
0	op+
H	000000
A	000000
0	
0	
B	000000
X	000000
0	
C	000000
Y	000000
	3
0	
0	

A table for an arithmetical/logical statement, + is choosen. 0 (has zero code).
LAB:+,H A,B(x),C(Y,3);
Notice that a word is 36 bits and a character 6 bits.

Example of a pass 1 result table unit.

In pass 1 it is easy to include the elimination of comments. This has beend done, and a kind of nested comments on different levels has been introduced. It is also possible to include a macro facility in OBM, and to take care of this during pass 1. This pass will also detect some syntactical errors, but further error checking is left to the next pass. If an error is found in a statement, a message is written and the rest of the state-ment will be skipped until ; .

5.2

THE SEGMENT TRANSLATOR

The result of the lexical analyzer is a sequence of statements in internal representation. The segment translator is called when the SEGMENT statement is found, and it is left when the END statement is found. The segment translator is divided into two parts, the data description phase and the instruction phase. The translator is in the data description phase until it encounters the first real instruction, then it is in the instruction phase. In the instruction phase a data description statement is illegal. The data description phase is building tables to be used in the instruction phase. The segment trans-lator is written as a number of subroutines, one for each possible OBM statement, and an administration routine.

5.2

Most of the instruction translator subroutines are written as a sequence of macro calls. This technique may be useful as a description of how the instructions shall be implemented, and as a description of the semantic meaning. It is also useful when moving the translator to another computer.

First an example of the routine used to translate the add statement will be given and discussed.

```

ADD*      . LABEL:4,TYP A1,A2,A3
          LABEL
          LOAD      ARO,2,L2
          TYP       0,TLO
          LENGHA    TLO,LO
          TRIM      ARO,L2,LO
          LOAD,2    AR1,3,L3
          TRIM,3    AR1,L3,LO
          ADDP      ARO,AR1,TLO
          LENGTH    1,L1          SYM1
          TRIM      ARO,LO,L1
          STORE     ARO,1          SYM1
          RETURN

```

The following described instructions are of course executed during translation. The instruction LABEL is used to take care of the label field if present. The label is given a value and put into the symbol table. The next instruction

```
LOAD ARO,2,L2
```

has the following meaning. Generate the object code to load the parameter described by the second parameter field into register ARO. Compute the bit length of the parameter and save it in L2. This is a rather complicated instruction, because of the flexible design of the parameters and their storage. The parameter may also be an indexed word.

The LOAD subroutine for UNIVAC contains 900 machine instructions, but the generated code for this load is in most cases only one instruction.

The next instruction TYP 0,TLO means that the type of the zeroth parameter field (it is TYP) is fetched and a pointer to it is saved in TLO. The instruction LENGHA TLO,LO is used to get the bit length from type TLO and save it in LO. The TRIM ARO,L2,LO is used to generate object code to trim the contents of register ARO from L2 bits to LO bits. In most cases LO is equal to L2 and no code is generated.

5.2

The next instruction is a LOAD instruction, but more complicated than the first one.

LOAD,2 AR1,3,L3

The instruction is used to generate code to load register AR1 by the third parameter. As it is possible to add directly from memory to ARO this load might be unnecessary, but the address must be used. The ,2 on the LOAD operator means that it is the second load, and that the generation of the instructions should be delayed until found necessary. During the execution of LOAD it is sometimes found that the code generation must be performed, and then the generation is not delayed.

The next instruction is a TRIM but the TRIM,3 means that it is a trimming of an operand whose load instructions might be delayed, and if trimming has to be done the load instructions must first be generated.

The instruction ADDP ARO,AR1,TLO generates object instructions to add the numbers in AR1 and ARO according to type TLO and give the result in ARO. If AR1 is not loaded and it is necessary to load the operand into AR1 these delayed instructions are put into the object code. In most cases the add instruction may operate on ARO and with the other operand in memory. Thus delay of code generation is very important for efficiency, even if there is some work to implement it.

After the addition we have to compute the length of the location where the result shall be stored. This is done by LENGTH 1,L1 SYM1 the length is stored in L1 and the entry to the symbol table is saved for future use in SYM1.

Now the ARO is stored in the location given by parameter one, and notice that the symbol table entry is already determined and saved in SYM1. The add statement object code generating subroutine is left by the RETURN instruction.

All the executable statements in OBM can be described in this way by a number of macro instructions. The code generating subroutines are actually described and constructed in this way, but this is not covered by this report. This is because of the size of this, and that these macro instructions are not constructed in the most systematic way.

5.2

The main concern of this work has not been the construction of this translator, but rather the definition of the OBM language. I hope that this translator construction will be studied in more detail in the future.

A small number of examples will now follow, but without comments.

IFEQ*

```

    LABEL
    TYP      0,TLO
    LOAD     ARO,1,L1
    TRIM     ARO,L1,L0
    LOAD,2   AR1,2,L2
    TRIM     AR1,L2,L0
    GETADR   LA0,9
    GETADR   LA1,10
    IFEQ     ARO,AR1,LA0,LA1 TLO
    RETURN

```

SETPTR*

```

    LABEL
    LOAD     ARO,2,L2
    TRIM     ARO,L2,(16)
    SPTR     ARO
    RETURN

```

EXIT*

```

    OSA,3
    RETURN

```

REPEAT*

```

    LABEL
    BLCK
    OPTION   'L'          REPL
    IND      3
    TAL,2    ARO,2,RP1
    LOAD,2   ARO,2,L2
    TRIM,2   ARO,L2,(16)
    RP1     .START       ARO
    TAL,3    ARO,3,RP2
    LOAD,3   ARO,3,L3
    TRIM,3   ARO,L3,(16)
    RP2     .STEP        ARO
    NEXT
    TAL,1    AR1,1,RP3
    LOAD,1   AR1,1,L1
    TRIM,1   AR1,L1,(16)
    RP3     .FIN         AR1
    RETURN
    REPL    TAL          ARO,1,RP4
    LOAD     ARO,1,L1
    TRIM     ARO,L1,(16)
    RP4     .LOOP        ARO
    RETURN

```

READ*

```

    LABEL
    RDWR
    TAL,2    ARO,3,LR1
    LOAD     ARO,3,L3
    TRIM     ARO,L3,(16)
    LR1     .ITEMS       ARO
    FORMAT
    RD

```


IMPLEMENTATION PROBLEMS

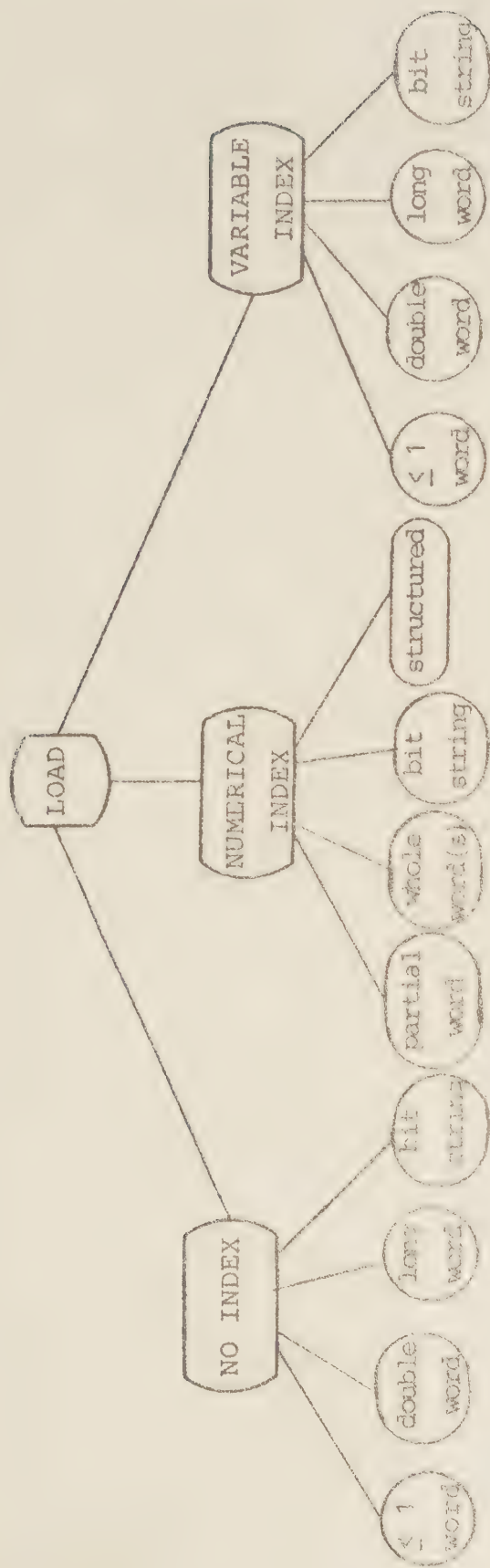
There are two types of problems, to produce efficient object code on the host computers for OBM statements and to construct OBM translators with reasonable size and efficiency for different computers. It is also important to write a translator in a machine independent way as far as possible, and thus to make the OBM language easy to implement on a number of computers.

One method is to break down the OBM statements to a sequence of lower instructions. These low level instructions can be regarded as a description and implementation language for OBM. Some examples of this are given in 5.2. In order to implement OBM we have to implement these low level instructions. Thus it is interesting to know if they are hard to implement or not. Some of these low level instructions are service instructions used by several OBM statements, but others are specific for one OBM statement. Most of the latter are simple to implement if we know what object code to generate. Some of the service routines are not used to generate code, but to build translator tables and to get information from such tables. These are easy to implement. Other service routines do generate object code, and are fundamental for OBM. The most used of these are LOAD, TRIM and STORE, they are used by almost all OBM statements. They are rather hard to implement, and on UNIVAC 900 instructions are used for LOAD, about 1300 for TRIM and more than 1100 for STORE, and in this text implementation they are not complete. LOAD and STORE have a common subroutine for indexes of about 1000 instructions. It is of course impossible to discuss in detail all the 900 instructions of the LOAD subroutine in this report, but the main ideas will be given. The subroutine can be regarded as a big tree structure, where the different branches are taken depending on the way of addressing and storage (see figure). This branching is rather simple and many of the branches are similar to each other. Thus the coding is, at least in theory, simple, but much code has to be produced. So far it is simple, but now the troubles begin. First the delayed code generation described in 5.2 makes it slightly more complicated. Then for indexed addressing we have to generate code for the loading of the index too. This is almost the same as performing another load, but some differences are present, for example indexing of an index is not permitted.

5.3

For this the same code as for the original load could be used, with some modifications. Another and harder problem is that some OBM statements must compute an address of a data word, without using the address for loading. This routine will be almost the same as the LOAD routine, but it will differ on a number of points. If a new routine for this has to be introduced in the translator, it means that the code for the central LOAD routine will be doubled, and the size of the translator is already a problem. Also from some other instructions, parts of the LOAD routine could be used to save code.

Thus we have a rather simple tree structure of the LOAD routine but then we may add another different structure on that tree to handle delayed code generation, loading of indices and address computations. And now the routine is not simple any more. This is a central problem of the OBM translator, to use a simple straightforward tree structure and thus make the programming simpler and the translator faster, or to add a more complicated structure and thus save space. To make the translator fast we may write the translator in a very space consuming way and then we may have to use a segmentation or paging technique, and the translator will be slow. There is a risk, to make the compiler fast, it may be slow.



The main structure of the load routine.

EXAMPLE

A printout from the test OBM translator for UNIVAC is presented as an example. Three programs and their relocatable segments are listed. The external data word descriptions are of special interest. The code is UNIVAC machine instructions and the lists are presented without comments.

00M.00M,S SETUP.PROGS,OFIL.
EOF

```
1 DE:SEGMENT L0; EXREF L2;  
2 T:TYP INTEGER,36; M:PODL,T 500;  
3 A:DATA M,0; B:DATA,T M(X),1,10; C:DATA M,100;  
4 X:DATA M,200;  
5 L0:IF=0,T A,L1; +,T A,A,B; L1:+=,T B,B; GO L2;  
6 END;  
7 DE1:SEGMENT; EXDEF L2;  
8 P:TYP INTEGER,36; MP:PODL,P 500,X;  
9 B:DATA MP,1;  
10 L2:--=,P B,B;  
11 ENL;  
12 PROGEND;
```

P OF SEGMENT ASSEMBLY TABLES >NO
P OF SEGMENT ASSEMBLY TABLES >NO
P OF PROGRAM SEGMENT TABLES >NO
T OF RELOCATABLE ELEMENT >YES

SEGMENT: DE START ADDRESS: 000000

EXREFD EMPTY

EXREFI

L2

EXDEFD

B 1 1 1 36 0 -2 498 000000000000

EXDEFI EMPTY

TEXT									RELINF
LC	ADDRESS	WORD	INSTRUCTION						
000	000001	0000000000012	00	00	00	00	0	000012	
006	000000	0000000000000	00	00	00	00	0	000000	
		0000000000001	00	00	00	00	0	000001	
001	000000	1001000000000	10	00	04	00	0	000000	LC RA 000
001	000001	7401000000007	74	00	04	00	0	000007	LC RA 001
001	000002	1001000000000	10	00	04	00	0	000000	LC RA 000
		1401000000001	14	00	04	00	0	000001	LC RA 000
001	000004	7464000000006	74	15	00	00	0	000006	LC RA 001
001	000005	7204002000000	72	01	00	00	1	000000	
001	000006	0101000000000	01	00	04	00	0	000000	LC RA 000
		1001000000001	10	00	04	00	0	000001	LC RA 000
001	000010	1201000000020	12	00	04	00	0	000020	
001	000011	0101000000001	01	00	04	00	0	000001	LC RA 000
001	000012	7420000000000	74	04	00	00	0	000000	LC RA 000001
001	000013	7204002000007	72	01	00	00	1	000007	

END OF TEXT

EXREFD

B 1 1 1 36 0 -2 498

EXREFI EMPTY

EXREFD EMPTY

EXREFI

11 000 01000000

TEXT

LC	ADDRESS	WORD	INSTRUCTION	RELINF
006	000000	000000000000	00 00 00 00 0 000000	
		000000000001	00 00 00 00 0 000001	
001	000000	100100000001	10 00 04 00 0 000001	LC RA 000001
001	000001	110100000020	11 00 04 00 0 000020	
001	000002	010100000001	01 00 04 00 0 000001	LC RA 000001
001	000003	720400200007	72 01 00 00 1 000007	

END OF TEXT

OF ELEMENT

00M.00M.5 SETUP,PROG1,OFIL.

1 OF

```

1  A:SEGMENT LO;
2  EXDEF L1,L2; EXREF LA1,LA2,LA3;
3  M:TYP INTEGER,36; HALV:TYP INTEGER,18; T:TYP INTEGER,7,E;
4  M:POOL,H 250; HM:POOL,HALV 250;
5  M1:DATA M,0; M2:DATA M,100; X:DATA M,201;
6  F1:DATA HM,0; F2:DATA HM,100; Y:DATA HM,1;
7  IF=0,H M,LO,L1; LO:AND,H M,F1,F2; L1:IF>0,HALV M2,LO,L2;
8  L3:OR,HALV M1,H1,F1; L2:IF>0,HALV F2(5),LA1,LO;
9  AND,I M1,M1,M2; IF=0,I F1,L1; IF=0,H M1(Y),LO;
10 OR,HALV F1,F2(X),M2;
11 END ;
12 END SEGMENT;
13 M:TYP INTEGER,36; M:POOL,H 4; S:DATA M,0; I:DATA M,1;
14 EIT:DATA,H M,2,1; HUNDRA:DATA,H M,3,100;
15 L1:H 1,I,EIT; +H S,S,I; IF=H S,HUNDRA,L1;
16 END;
17 B:SEGMENT ;
18 EXDEF LA1,LA2; EXREF L2,U3;
19 P:TYP INTEGER,36; M:POOL,P 100;
20 R1:DATA M,0; Q2:DATA M,99;
21 L1:OR,P Q1,L1; IF=0,P Q3,LA1,LA2; LA1:OR,P Q1,Q1,Q2;
22 LA2:XOR,P Q2,Q1,Q2; IF>0,P Q1,U3,LA1;
23 END ;
24 PROGCEND;

```

MP OF SEGMENT ASSEMBLY TABLES >NO

MP OF SEGMENT ASSEMBLY TABLES >NO

MP OF SEGMENT ASSEMBLY TABLES >NO

MP OF SEGMENT ASSEMBLY TABLES >NO

MP OF SEGMENT ASSEMBLY TABLES >NO

SEGMENT: A

START ADDRESS: 000003

EXREFD EMPTY

EXREFI

LA1
LA2
LA3

EXDEFD EMPTY

EXDEFI

L1 000001000014
L2 000001000030

TEXT

LC	ADDRESS	WORD	INSTRUCTION	RELINF
006	000000	000000000000	00 00 00 00 0 000000	
		000000000001	00 00 00 00 0 000001	
001	000000	100100000000	10 00 04 00 0 000000	LC RA 000
001	000001	740100000003	74 00 04 00 0 000003	LC RA 001
001	000002	742000000014	74 04 00 00 0 000014	LC RA 001
001	000003	101100000372	10 02 04 00 0 000372	LC RA 000
001	000004	735100000022	73 12 04 00 0 000022	
		732100000022	73 04 04 00 0 000022	
001	000006	101100000454	10 02 07 00 0 000454	LC RA 000
001	000007	735100000372	73 12 07 00 0 000022	
		732100000022	73 04 07 00 0 000022	
		420100000023	42 00 04 00 0 000023	
		735000000044	73 13 04 00 0 000044	
001	000013	010100000000	01 00 04 00 0 000000	LC RA 000
		100100000144	10 00 04 00 0 000144	LC RA 000
006	000002	000000400000	00 00 00 00 2 000000	
001	000015	450100000002	45 00 04 00 0 000002	LC RA 006
001	000016	740500000003	74 01 04 00 0 000003	LC RA 001
001	000017	742000000030	74 04 00 00 0 000030	LC RA 001
001	000020	100100000000	10 00 04 00 0 000000	LC RA 000
006	000003	000000777777	00 00 00 00 3 177777	
001	000021	100000000003	10 00 03 00 0 000003	LC RA 006
001	000022	420000000020	42 00 03 00 0 000020	
001	000023	401100000372	40 02 04 00 0 000372	LC RA 000
001	000024	735000000044	73 13 04 00 0 000044	
		735100000022	73 12 04 00 0 000022	
		732100000022	73 04 04 00 0 000022	
001	000027	010100000000	01 00 04 00 0 000000	LC RA 000
		100500000456	10 01 04 00 0 000456	LC RA 000
001	000031	735100000022	73 12 04 00 0 000022	
		732100000022	73 04 04 00 0 000022	
001	000033	740500000000	74 01 04 00 0 000000	LC RA 000001
001	000034	742000000003	74 04 00 00 0 000003	LC RA 001
001	000035	100100000000	10 00 04 00 0 000000	LC RA 000
006	000004	000000000177	00 00 00 00 0 000177	

LC RA 000

LC RA 000

LC RA 000

LC RA 000

LC RA 000

LC RA 001

LC RA 000

LC RA 000

LC RA 001

LC RA 000

LC RA 000

LC RA 000

LC RA 000

LC RA 000

END OF TEXT

SEGMENT: BER

START ADDRESS: 777777

EXREFD EMPTY

EXREFI EMPTY

EXLEFD EMPTY

EXDEFI EMPTY

LC	TEXT ADDRESS	WORD	INSTRUCTION	RELINF
000	000002	0000000000001	00 00 00 00 0 000001	
		0000000000144	00 00 00 00 0 000144	
006	000000	0000000000000	00 00 00 00 0 000000	
		0000000000001	00 00 00 00 0 000001	
001	000000	1001000000001	10 00 04 00 0 000001	LC RA 000
		1401000000002	14 00 04 00 0 000002	LC RA 000
001	000002	7444000000004	74 15 00 00 0 000004	LC RA 001
001	000003	7204002000000	72 01 00 00 1 000000	
001	000004	0101000000001	01 00 04 00 0 000001	LC RA 000
		1001000000000	10 00 04 00 0 000000	LC RA 000
		1401000000001	14 00 04 00 0 000001	LC RA 000
001	000007	7444000000011	74 15 00 00 0 000011	LC RA 001
001	000010	7204002000000	72 01 00 00 1 000000	
001	000011	0101000000000	01 00 04 00 0 000000	LC RA 000
		1001000000000	10 00 04 00 0 000000	LC RA 000
		5401000000003	54 00 04 00 0 000003	LC RA 000
001	000014	7420000000000	74 04 00 00 0 000000	LC RA 001
001	000015	7204002000007	72 01 00 00 1 000007	

END OF TEXT

SEGMENT: B

START ADDRESS: 777777

EXDEFB EMPTY

EXDEFB

L2
L3

EXDEFB EMPTY

EXDEFB

L01 000001000000
L02 000001000011

LC	TEXT ADDRESS	WORD	INSTRUCTION	RELINF
006	000000	0000000000000	00 00 00 00 0 000000	
		0000000000001	00 00 00 00 0 000001	
001	000000	1001000000000	10 00 04 00 0 000000	LC RA 000
001	000001	7401000000000	74 00 04 00 0 000000	LC RA 000001
001	000002	1001000000143	10 00 04 00 0 000143	LC RA 000
001	000003	7401000000005	74 00 04 00 0 000005	LC RA 001
001	000004	7420000000011	74 04 00 00 0 000011	LC RA 001
001	000005	1001000000000	10 00 04 00 0 000000	LC RA 000
		4001000000143	40 00 04 00 0 000143	LC RA 000
001	000007	7335000000044	73 13 04 00 0 000044	
001	000010	0101000000000	01 00 04 00 0 000000	LC RA 000
		1001000000000	10 00 04 00 0 000000	LC RA 000
		4101000000143	41 00 04 00 0 000143	LC RA 000
001	000013	7335000000044	73 13 04 00 0 000044	
001	000014	0101000000143	01 00 04 00 0 000143	LC RA 000
		1001000000000	10 00 04 00 0 000000	LC RA 000
001	000016	7415000000005	74 03 04 00 0 000005	LC RA 001
001	000017	7405000000000	74 01 04 00 0 000000	LC RA 000002
001	000020	7420000000005	74 04 00 00 0 000005	LC RA 001
001	000021	7204002000007	72 01 00 00 1 000007	

END OF TEXT

000M.00M,S SETUP.PROCESS OF IL.
0000F

```

1      .OP:SEGMENT L1;
2      A:IF 1,0,0,0,3,0,N,S,M:POOL,H,20;
3      A:DATA M,0; B:DATA,H M,1,5;
4      L1:IF H A,A,B; IF 0,0,H A,L1,L2,L3,L4,H B,A,B;
5      END: PROGEND;

```

DUMP OF SEGMENT ASSEMBLY TABLES >NO

DUMP OF PROGRAM SEGMENT TABLES >NO

LIST OF RELOCATABLE ELEMENT >YES

SEGMENT: OP

START ADDRESS: 000000

EXREFD EMPTY

EXREFI EMPTY

EXDEFD EMPTY

EXDEFI EMPTY

LC	TEXT ADDRESS	WORD	INSTRUCTION	RELINF
000	000001	00000000000005	00 00 00 00 0 000005	
006	000000	00000000000000	00 00 00 00 0 000000	
		00000000000001	00 00 00 00 0 000001	
001	000000	10010000000000	10 00 04 00 0 000000	LC RA 000
		10010000000001	10 00 07 00 0 000001	LC RA 000
001	000002	74566000000000	74 13 13 00 0 000000	OP 15 0,0
001	000003	01010000000000	01 00 04 00 0 000000	LC RA 000
		10010000000000	10 00 04 00 0 000000	LC RA 000
001	000005	73510000000004	73 12 04 00 0 000004	
001	000006	74110000000012	74 02 04 00 0 000012	LC RA 001
001	000007	73510000000001	73 12 04 00 0 000001	
001	000010	74010000000012	74 00 04 00 0 000012	LC RA 001
001	000011	74200000000000	74 04 00 00 0 000000	LC RA 001
001	000012	10010000000000	10 00 04 00 0 000000	LC RA 000
		10010000000001	10 00 07 00 0 000001	LC RA 000
001	000014	74566000000000	74 13 13 00 0 000000	OP 15 0,0
001	000015	01010000000001	01 00 04 00 0 000001	LC RA 000
001	000016	72040020000007	72 01 00 00 1 000007	

END OF TEXT

OPERATOR SUBROUTINE TEXT

OPCODE: 15 TYPE: 1 0 1 32 0 0 0 0 0
 SIZE: 15 SIZES: 3

010	000000	020000000000	02 00 00 00 0 000000	
007	000000	450100000000	45 00 04 00 0 000000	LC RA 010
007	000001	742000000004	74 04 00 00 0 000004	LC RA 007
010	000001	757777777777	75 17 17 17 3 177777	
007	000002	410100000001	41 00 04 00 0 000001	LC RA 010
007	000003	100100000021	10 00 04 00 0 000021	
007	000004	450457777764	45 01 02 17 3 177764	LC RA 010
007	000005	742000000010	74 04 00 00 0 000010	LC RA 007
007	000006	410457777765	41 01 02 17 3 177765	LC RA 010
007	000007	100460000010	10 01 03 00 0 000010	
		140100000023	14 00 04 00 0 000023	
010	000002	420000000000	42 00 00 00 0 000000	
007	000011	440100000002	44 00 04 00 0 000002	LC RA 010
007	000012	720400200000	72 01 00 00 1 000000	
		741113000000	74 02 04 13 0 000000	
007	000014	410100000001	41 00 04 00 0 000001	LC RA 010
007	000015	100100000021	10 00 04 00 0 000021	
		742013000000	74 04 00 13 0 000000	

END OF OPERATOR SUBROUTINE TEXT

END OF ELEMENT

ACKNOWLEDGEMENT

I wish to thank Eric Astor for his remarks on this paper.

REFERENCE

- [1] Göran Fries OBM- A Machine Independent Machine Oriented
Language: CODEN: LUNFD6/(NFIP-3005)/1-83/(1978).
Department of Computer Sciences, Lund University.

